List comprehensions in Common Lisp

My favorite programming languages are undoubtedly *Lisp*s, originally Scheme but todays I enjoy a lot Common Lisp and (a little bit less but still fairly enough) Clojure. I am however quite a polyglot programmer: I taught (and thus used, a lot) Java for several years. And at work I program a lot in Ocaml and Python for work.

There's no programming language that I really hate (except VB) or love (except Lisp). Let's take for example Python: not efficient, full of what I think are ill-designed constructs (e.g. `lambda`, also the lexical bindings) but clearly usable. One feature I use a lot in Python is the *comprehensions*. There are comprehension expressions for building lists, sets, dictionaries and (perhaps most interestingly) generators. Today I will only discuss the case of *list comprehensions* because ... I think that's an interesting topic.

A simple list comprehension expression is written as follows in Python:

```
[ <expr> for <var> in <list> ]
```

This reads: "build the list of in which the occurrences of are replaced in turn by the successive elements of the source ."

Note that the source of the comprehension can be something else than a list (it can be an *iterable* or a generator) but we will mostly consider the case for lists (although we will make some remarks about this aspect at the end).

Remark: the functional programming language Haskell has a similar syntactic construct, and so does Clojure.

Let's see some examples of Python comprehensions.

```
>>> [i*i for i in [1, 2, 3, 4, 5]]
[1, 4, 9, 16, 25]
```

Comprehensions can also nest, performing a kind of *cartesian product*, e.g.:

```
>>> [(i, j) for i in [1, 2, 3, 4]
             for j in ['A', 'B']]
[(1, 'A'), (1, 'B'), (2, 'A'), (2, 'B'), (3, 'A'), (3, 'B'), (4, 'A'), (4, 'B')]
```

It is also possible to filter the elements:

```
>>> [i for i in [1, 2, 3, 4, 5, 6, 7, 8] if i % 2 == 0]
[2, 4, 6, 8]
```

You know that already, but Common Lisp does not provide *list comprehension expressions*. However, the programmable programming language is never short of macro-based solutions for "missing" constructs!

So, our objective will be to find a few ways to provide list comprehensions to Common Lisp.

The list monad

No no no! This is not a tutorial about monads! But please don't hate monads or other interesting design patterns of functional programming languages. There's some beauty in the concept, and I will try to convey this beauty by considering one of the simplest case: the *list monad*.

The fundamental operator that we are concerned with is a function that does not look as useful as we might think: `append-map`.

```
(defun append-map (f ll)
  (if (endp ll)
      '()
      (append (funcall f (car ll)) (append-map f (cdr ll)))))
```

Of course, `append` is interesting, and `map`/`mapcar` too, but mixing the two does not look extremely useful. Alghouth this works, for example:

```
(append-map (lambda (x) (list x (* 10 x))) '(1 2 3 4 5))

=> (1 10 2 20 3 30 4 40 5 50)
```

The type of this function is something like `(a -> L a) * L a -> L a` where we interpret `L a` as "a List of `a`'s". Not that I want to for you to think in types, but we need at least an intuitive understanding of the types of our functions, right ?

The first and most famous monadic combinator is called `bind` and in the case of the list monad it is almost a synonymous for `append-map`.

```
;; L a * (a -> L a) -> L a
(defun list-bind (l f)
  (append-map f l))
```

Remark: for a monad `M` the type of `bind` is `M a * (a -> M a) -> M a`. It's good to know even if we will only consider the list monad `L`.

The second fundamental combinator is `return` that is the simplest function we can imagine with a type `a -> L a` (`a -> M a` for a monad `M` in general).

```
;; a -> L a
(defun list-return (x)
  (list x))
```

And we in fact already have everything we need to write simple comprehensions.

Let's start slowly:

```
(list-bind '(1 2 3 4 5)
           (lambda (x) (list-return (* x x))))

=> (1 4 9 16 25)
```

And now something a little bit more involved.

```
(list-bind '(1 2 3 4)
           (lambda (x) (list-bind '(A B)
                                  (lambda (y) (list-return (cons x y))))))

=> ((1 . A) (1 . B) (2 . A) (2 . B) (3 . A) (3 . B) (4 . A) (4 . B) (5 . A) (5 . B))
```

If we want to add some condition in the comprehension, we need a third combinator called `fail`. In the case of the list monad, failing simply boils down to yielding the empty list.

```
(defun list-fail ()
  (list))
```

And now we can write e.g.:

```
(list-bind '(1 2 3 4 5 6 7 8)
           (lambda (x) (if (evenp x) (list-return x) (list-fail))))

=> (2 4 6 8)

(list-bind '(1 2 3 4 5 6 7 8)
           (lambda (x) (if (oddp x) (list-return x) (list-fail))))

=> (1 3 5 7)
```

The do notation

If we compare to the comprehension expressions of Python or Haskell, we can probably say that using directly the combinators `bind`, `return` and `fail` (specialized for lists) is more explicit but less elegant.

Haskell provides a `do` notation that greatly improves the monadic combinations. In fact, monads (and their relatives functors and applicatives) are mostly about plumbing, as we can see with the examples above. The do notation allows to hide and reorganize some of this plumbing.

This is a straightforward macro in Common Lisp.

```
;; a permissive equality for keywords in constructs (e.g. when or :when)
(defun eq-keyword (kw str)
  (and (symbolp kw)
       (string= (symbol-name kw) str)))

(defmacro do-list (&body body)
  (cond
    ((null body) (progn))
    ((null (cdr body)) (car body))
    ((eq-keyword (cadr body) "<-")
     `(list-bind ,(caddr body) (lambda (,(car body)) (do-list ,@(cdddr body)))))
    ((eq-keyword (car body) "WHEN")
     `(if ,(cadr body) (do-list ,@(cddr body)) (list-fail)))
    ((eq-keyword (car body) "YIELD")
     `(list-return ,(cadr body)))
    (t (error "Not a do-able expression: ~S" `(quote ,body)))))
```

And now we can write:

```
(do-list
  i <- '(1 2 3 4 5)
  yield (* i i))

=> (1 4 9 16 25)

(do-list
  i <- '(1 2 3 4)
  j <- '(A B)
  yield (cons i j))

=> ((1 . A) (1 . B) (2 . A) (2 . B) (3 . A) (3 . B) (4 . A) (4 . B))
```

```
(do-list
  i <- '(1 2 3 4 5 6 7 8)
  when (evenp i)
  yield i)

=> (2 4 6 8)

(do-list
  i <- '(1 2 3 4 5 6 7 8)
  when (oddp i)
  yield i)

=> (1 3 5 7)
```

All is well, we have now a simple macro for list comprehensions, and as you can see the monadic thinking make things very concise and (if you spend some time to understand) straightforward.

But is this the simplest and best way to bring list comprehensions to Lisp?

Let's see another (and please be reassured "un-monadic") way...

The Loop way to comprehensions

Instead of building on basic functions – the Monad way – we can build our list comprehensions on higher-level abstractions. In plain Common Lisp, the `loop` macro (should be called the `loop` DSL) is a good candidate. For `loop`-haters I would recommend porting the code below to the more Lispy and undoubtedly (even) more powerful `iterate` macro (that you'll find on quicklisp of course). But let's stick with `loop` that already provides everything we need (and more !).

So, how would we write the list of squares in loop ? Easy !

```
(loop for i in '(1 2 3 4 5)
   collect (* i i))

=> (1 4 9 16 25)
```

For the nested comprehensions, the `append` clause is our friend.

```
(loop for i in '(1 2 3 4)
   append (loop for j in '(A B)
             collect (cons i j)))
```

```
=> ((1 . A) (1 . B) (2 . A) (2 . B) (3 . A) (3 . B) (4 . A) (4 . B))
```

Filtering is also easy with the `when` clause.

```
(loop for i in '(1 2 3 4 5 6 7 8)
   when (evenp i)
   collect i)

=> (2 4 6 8)
```

By looking at these examples, we might say that list comprehension expressions are not really need in Common Lisp since the corresponding `loop` expressions are both readable and efficient (probably more so than using the list monad).

However, the loop syntax is complex and it is still useful to provide a simpler abstraction for list comprehensions. Moreover, we can exploit various loop features to enrich our comprehension framework.

The `list-of` macro for list comprehensions

We now reach the final stage of our exploration of list comprehensions. We will build a (relatively) simple macro named `list-of` that will macroexpand to `loop` expressions.

The basic comprehensions will be of the form:

```
(list-of <expr> for <var> in <list>)
```

More generally, all `list-of` expressions will be of the form `(list-of <expr> for <var> ...)`. Hence, the first clause will always be a `loop`-like `for` clause.

The following transformer manages to extract this first clause from a `list-of` expression.

```
(defun transform-first-clause (expr)
  (cond
    ((null expr) (error "Empty comprehension: missing first 'for' clause"))
    ((eq-keyword (car expr) "FOR")
     (values `(for ,(cadr expr) ,(caddr expr) ,(cadddr expr)) (cddddr expr)))
    (t (error "First 'for' clause missing in comprehension."))))

(transform-first-clause '(for i in '(1 2 3 4 5) for j in '(A B) for k across "Hello"))

=> (FOR I IN '(1 2 3 4 5))     ; the extracted clause as a first value
=> (FOR J IN '(A B) FOR K ACROSS "Hello")  ; the remaining clauses as a second value
```

Now we write a second transformer for all the remaining clauses. The supported clauses are the following ones:

- the `for` clause for nested comprehensions (using an inner `loop` nested within an `append` clause)
- the `and` clause for parallel comprehensions (similarly to multiple `for` clauses in a single `loop`)
- the `with` clause to fix a temporary variable (`with <var> = <expr>` corresponds to `loop`'s `for <var> = <expr>`)
- other intresting `loop` clauses: `when` for filtering, `until` for early terminations, and `with` for temporary variables.

Note that this transformer is not very robust (with many cadd...r's), but adding all the error cases would make the code less readable I guess. For the real thing I would strongly recommand using the `optima` pattern matcher.

```
(defun transform-clause (expr)
  (if (null expr)
      (values :END '() '())
      (cond
        ((eq-keyword (car expr) "AND")
         (values :AND `(for ,(cadr expr) ,(caddr expr) ,(cadddr expr)) (cddddr expr)))
        ((eq-keyword (car expr) "WHEN")
         (values :WHEN `(when ,(cadr expr)) (cddr expr)))
        ((eq-keyword (car expr) "UNTIL")
         (values :UNTIL `(until ,(cadr expr)) (cddr expr)))
        ((eq-keyword (car expr) "WITH")
         (values :WITH `(for ,(cadr expr) ,(caddr expr) ,(cadddr expr)) (cddddr expr)))
        ((eq-keyword (car expr) "FOR")
         (values :FOR `(for ,(cadr expr) ,(caddr expr) ,(cadddr expr)) (cddddr expr)))
        (t (error "Expecting 'and', 'for', 'when' or 'until' in comprehension.")))))

(transform-clause '())

=> :END ,  NIL,  NIL

(transform-clause '(and i in '(1 2 3 4) for j in '(A B) collect i))

=> :AND                   ; the kind of clause as a first value
=> (FOR I IN '(1 2 3 4))  ; the clause content as a second value
=> (FOR J IN '(A B) COLLECT I)  ; the remaining clauses as a last value
```

Now we can write the main transformer, that simply recurse on the clause transformer above. The case for the nested comprehensions injects the *loop-within-append* expression. But everything is otherwise rather straightforward.

```
(defun list-of-transformer (what expr)
  (multiple-value-bind (kind next rexpr)
      (transform-clause expr)
    (case kind
      (:END `(collect ,what))
      ((:AND :WHEN :UNTIL :WITH) (append next (list-of-transformer what rexpr)))
      (:FOR `(append (loop ,@next ,@(list-of-transformer what rexpr)))))))

(list-of-transformer '(cons i j) '(with k = i and i in '(1 2 3 4) for j in '(A B)))

=> (FOR K = I FOR I IN '(1 2 3 4) APPEND
     (LOOP FOR J IN '(A B)
           COLLECT (CONS I J)))
```

The macro itself is simply the construction of a `loop` expression with a body generated by our two transformer `transform-first-clause` and `list-of-transformer`.

```
(defmacro list-of (what &body body)
  (multiple-value-bind (first-clause rest)
      (transform-first-clause body)
    `(loop ,@first-clause ,@(list-of-transformer what rest))))
```

Now let's see somme expansions... we'll also check that things are working (at least seemingly) well.

```
(macroexpand-1 '(list-of (* i i) for i in '(1 2 3 4 5)))

=> (LOOP FOR I IN '(1 2 3 4 5)
         COLLECT (* I I))

(list-of (* i i) for i in '(1 2 3 4 5))

=> (1 4 9 16 25)
```

Let's try the `with` clause (and see that it is expanded to a `for` clause in the loop).

```
(macroexpand-1 '(list-of k
                 for i in '(1 2 3 4 5)
                 and j in '(1 2 3 4 5) with k = (+ i j)))

=> (LOOP FOR I IN '(1 2 3 4 5)
         FOR J IN '(1 2 3 4 5)
         FOR K = (+ I J)
         COLLECT K)

(list-of k for i in '(1 2 3 4 5)
           and j in '(1 2 3 4 5)
           with k = (+ i j))

=> (2 4 6 8 10)
```

Filtering with `when` of course works.

```
(list-of i
  for i in '(1 2 3 4 5 6 7 8)
  when (evenp i))

=> (2 4 6 8)

(list-of i
  for i in '(1 2 3 4 5 6 7 8)
  when (oddp i))

=> (1 3 5 7)
```

The nesting of comprehensions yields nested loops as expected.

```
(macroexpand-1 '(list-of (cons i j)
                 for i in '(1 2 3 4)
                 for j in '(A B)))

=> (LOOP FOR I IN '(1 2 3 4)
         APPEND (LOOP FOR J IN '(A B)
                      COLLECT (CONS I J)))

(list-of (cons i j)
  for i in '(1 2 3 4)
  for j in '(A B))
```

```
=> ((1 . A) (1 . B) (2 . A) (2 . B) (3 . A) (3 . B) (4 . A) (4 . B))
```

To illustrate the `until` clause let's stop a little bit earlier.

```
(list-of (cons i j)
  for i in '(1 2 3 4)
  until (= i 3)
  for j in '(A B))

=> ((1 . A) (1 . B) (2 . A) (2 . B))
```

It is to be compared with parallel comprehensions, as in the following example.

```
(macroexpand-1 '(list-of (cons i j)
                 for i in '(1 2 3 4)
                 and j in '(A B)))

=> (LOOP FOR I IN '(1 2 3 4)
         FOR J IN '(A B)
         COLLECT (CONS I J))

(list-of (cons i j)
  for i in '(1 2 3 4)
  and j in '(A B))

=> ((1 . A) (2 . B))
```

And the last one is left as en excercise.

```
(list-of (list i j k)
  for i in '(1 2 3 4 5)
  and j in '(A B C D E)
  when (oddp i)
  for k in '(1 2 3 4)
  until (= i 5)
  when (evenp k))
```

Conclusion

So what did we achieve ?

First, we found two different ways to provide *list comprehensions* in Common Lisp. Whereas in many programming languages this is a matter of hope or faith, a Lisp programmer can take his destiny at hand and introduce the feature he wants *the way* he wants it !

Another take away is that monadic thinking is useful, even if you must be able to go beyond the type-based pluming they propose... In a way monads are a kind of well-typed-although-limited macros. Simply relying on the powerful `loop` mini-language allowed us to go beyond simple comprehensions (especially with the `until` clause).

If compared to comprehension expressions found in other languages, it is not a bad start. Thanks to `loop` we can take different sources: lists (with `in`), strings (with `across`), etc. There would be some (simple) work to support more collections such as hashtables. Perhaps a better thing would be to base our macro on `iterate` since the latter is extensible.

Finally, Python also has set, dictionnary comprehensions as well as generator expressions (actually closer to Clojure's sequence comprehensions). This goes beyond our `list-of` macro but it is a topic I do intend to further study.

And that's it for today...